2장 정렬 알고리즘

목차

기본 개념(1)

- 여러 개의 원소로 구성된 리스트가 주어졌을 때, 이 원소들을 순서대로 재배치하는 일
- 용어
  - 레코드(record) : 정렬할 각 원소
  - 필드(field) : 레코드에 포함되어 있는 여러 가지 정보
  - 키(key) : 레코드를 대표하며, 레코드 간의 순서를 나타내는 자료
- 종류
  - 내부(internal) 정렬 : 주 기억 장치에 정렬할 레코드가 있음
  - 외부(external) 정렬 : 보조 기억 장치에 정렬할 레코드가 있음
- 수행 시간
  - 기초적인 방법 : N^2
  - 향상된 방법 : $N \log N$ or $N^{3/2}$

기본 개념(2)

- 안정성(stability)
  - 같은 키 값을 가지는 레코드의 상대적인 위치가 유지되면 안정적(stable)이라고 함
  - 예 : 학생 리스트를 성명순으로 정렬할 때 이름이 같은 학생들의 상대적인 순서가 그대로 유지되면 안정적임
  - 간단한 방법은 주로 안정적임
- 공간
  - 제자리(in place) 정렬 알고리즘 : 입력 배열 이외의 추가 기억장소의 수가 상수 개를 넘지 않음
  - 연결 리스트 표현 : 리스트 포인터를 위한 N 개의 추가 기억장소 필요
  - 복사본 필요 : 정렬시킬 배열과 동일한 크기의 추가 기억장소 필요

기본 개념(3)

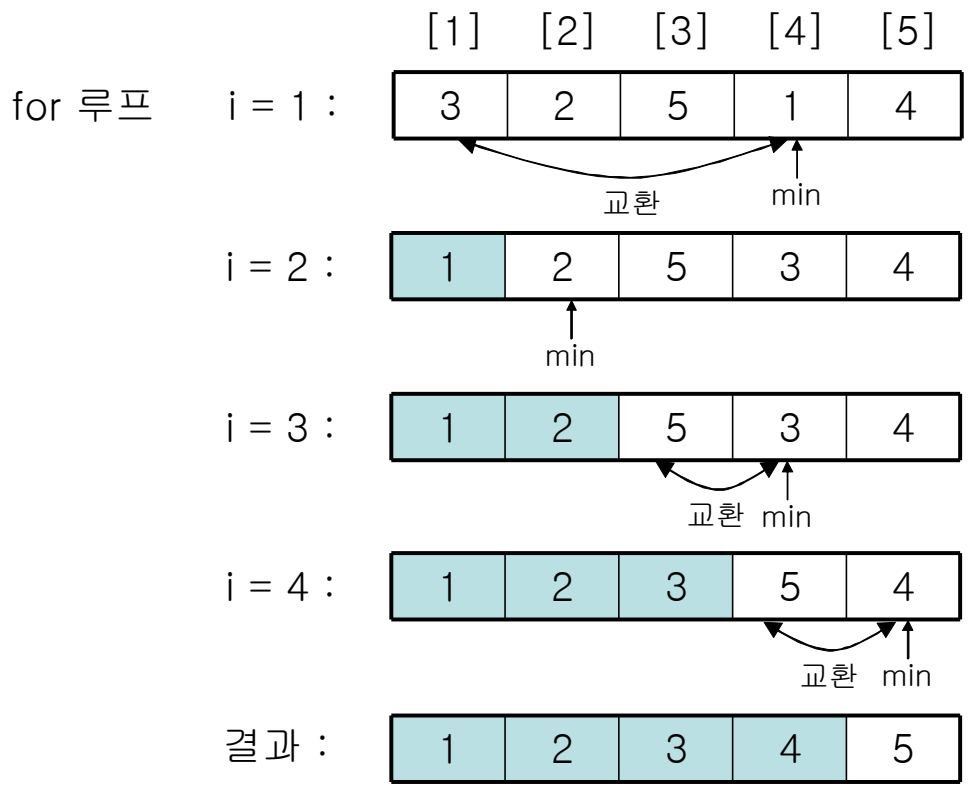
- 배열 사용
  - 정수 배열 a[N+1]
  - N+1 개의 배열을 선언
  - a[0] (sentinel key 혹은 dummy key)
  - a[1]~ a[N] N 개의 데이터를 저장
- 난수 발생
  - 실행 시간의 측정을 위해 사용
  - random 모듈의 randint() 메소드와 shuffle() 메소드 사용
- 시간 측정
  - time 모듈의 time() 메소드 사용

선택 정렬 (selection sort)

- 배열에서 가장 작은 원소를 찾아 첫 번째 원소와 교환하고 두 번째 작은 원소를 찾아 두 번째 원소와 교환하고 이러한 방식으로 전체가 정렬될 때까지 계속함
- 특징
  - 레코드가 실제로 교환되는 것은 많아야 한번 뿐이므로 작은 키와 매우 큰 레코드를 가지는 화일을 정렬하는데 적합함
  - 실행 시간은 입력 자료의 순서에 민감하지 않음
  - 제자리 정렬
  - 불안정적

수행 과정

- a[] = (3, 2, 5, 1, 4)


[1] [2] [3] [4] [5]
for 루프
i = 1 :
3 2 5 1 4
교환
min
i = 2 :
1 2 5 3 4
min
i = 3 :
1 2 5 3 4
교환 min
i = 4 :
1 2 3 5 4
교환 min
결과 :
1 2 3 4 5

성능 특성

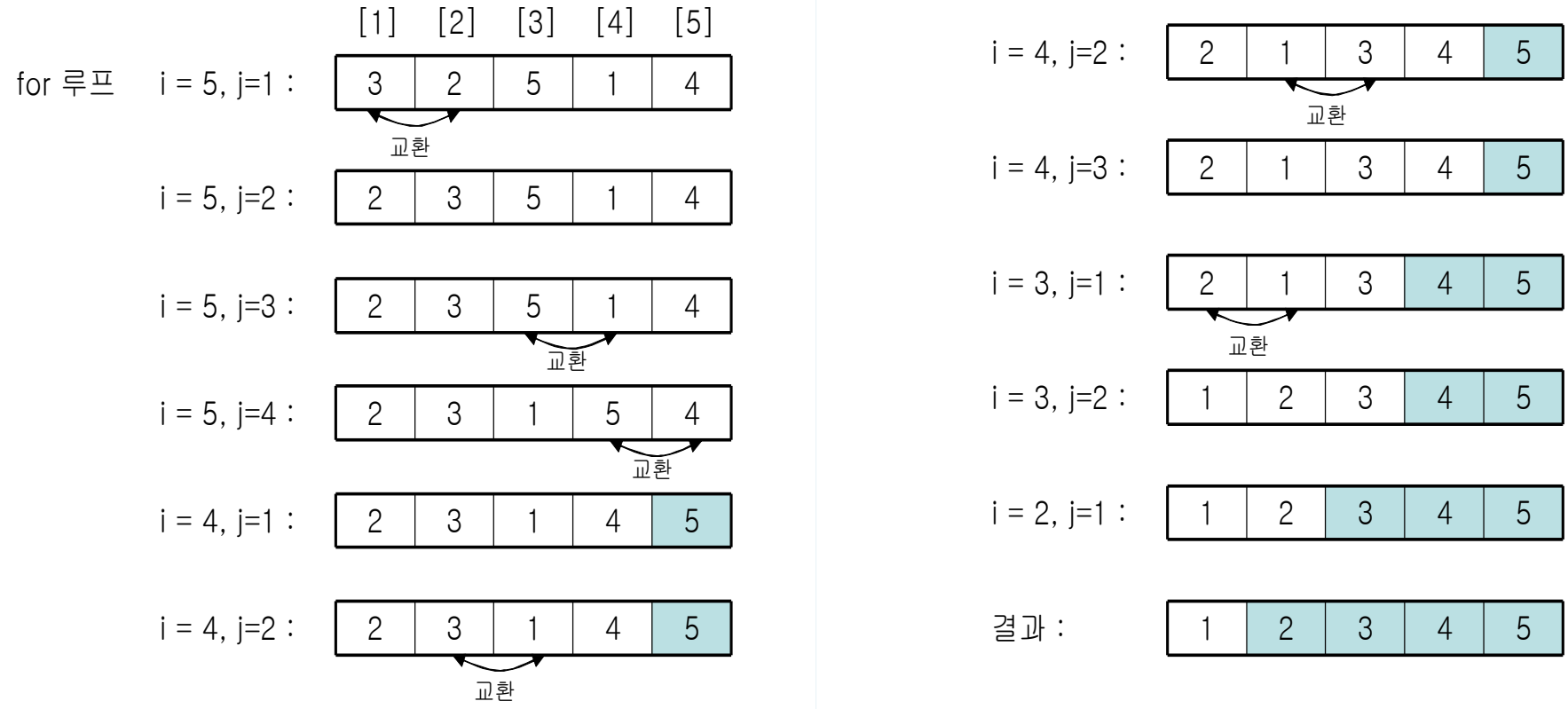
- N 개의 원소 각각에 대해 $N-1$ 번의 비교
- 전체 비교 횟수 $N(N-1)/2$
- 전체 시간 복잡도 $O(N^2)$
- 큰 레코드와 작은 키를 가지는 화일의 경우 효율적임

버블 정렬 (bubble sort)

- 마치 거품이 물 위로 올라가는 것 같이 루프를 한번 반복할 때 마다 가장 큰 값을 가진 원소가 가장 뒤쪽으로 이동함
- 특징
  - 레코드를 계속 교환하므로 레코드의 크기가 큰 경우에 불리
  - 거의 정렬이 된(almost sorted) 화일인 경우 유리
  - 안정적인 제자리 정렬

수행 과정

◆ a[] = (3, 2, 5, 1, 4)


[1] [2] [3] [4] [5]
for 루프 i = 5, j=1 : 3 2 5 1 4
교환
i = 5, j=2 : 2 3 5 1 4
i = 5, j=3 : 2 3 5 1 4
교환
i = 5, j=4 : 2 3 1 5 4
교환
i = 4, j=1 : 2 3 1 4 5
i = 4, j=2 : 2 3 1 4 5
교환
i = 4, j=2 : 2 1 3 4 5
교환
i = 4, j=3 : 2 1 3 4 5
i = 3, j=1 : 2 1 3 4 5
교환
i = 3, j=2 : 1 2 3 4 5
i = 2, j=1 : 1 2 3 4 5
결과 : 1 2 3 4 5

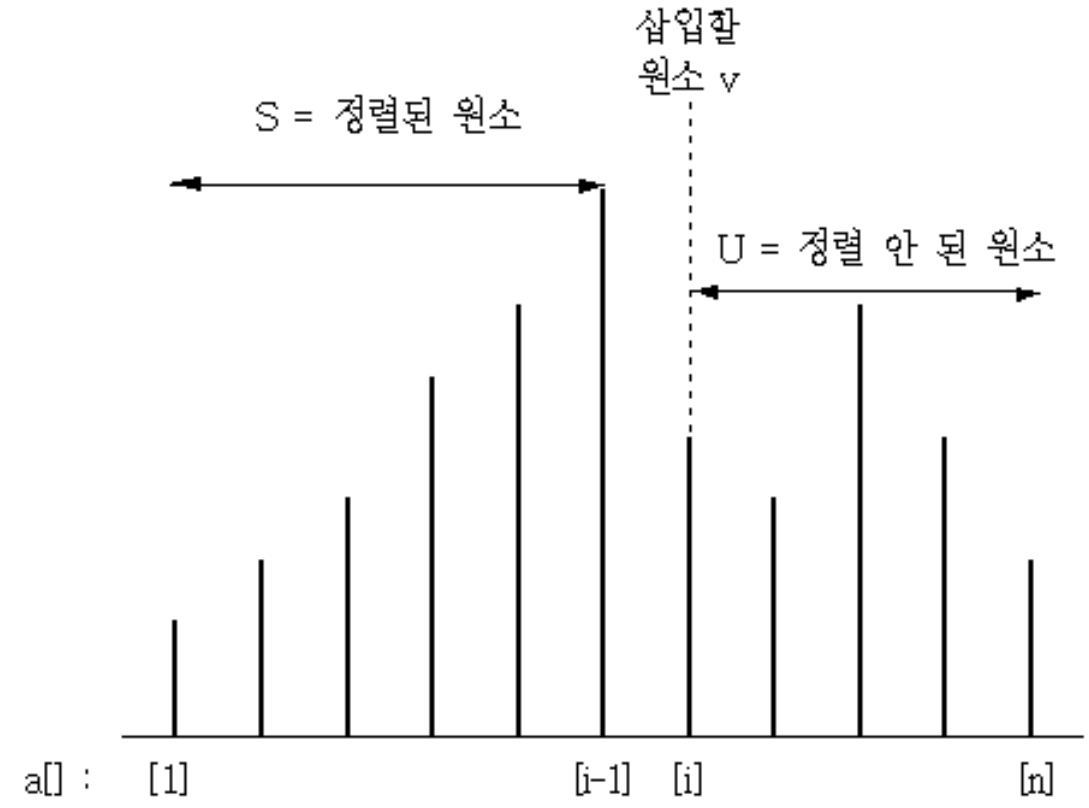
성능 특성

- N 개의 원소 각각에 대해 $N-1$ 번의 비교
- 전체 비교 횟수 $N(N-1)/2$
- 전체 시간 복잡도 $O(N^2)$

삽입 정렬 (insertion sort)

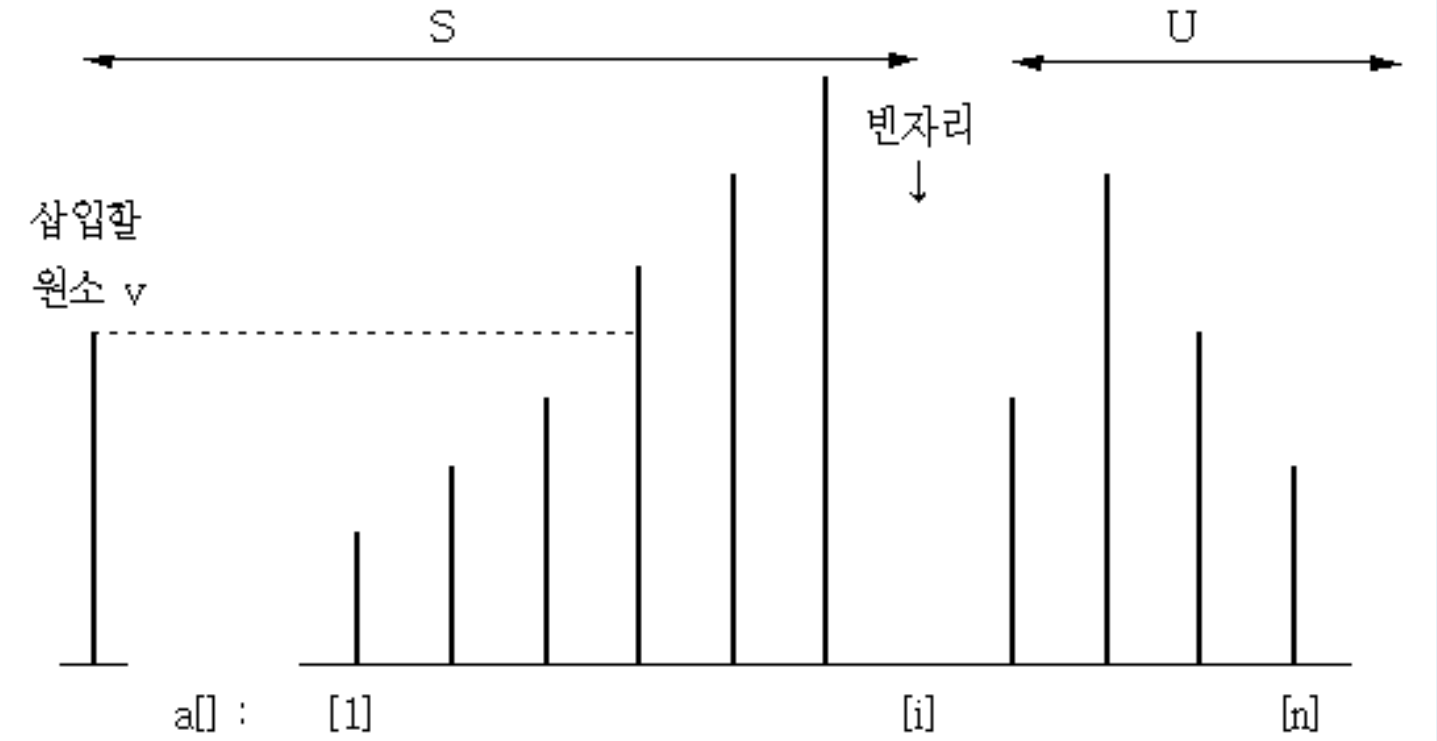
- 카드 놀이를 할 때 손에 들고 있는 카드를 정렬하는 것과 유사
- 오른쪽으로 움직이며 차례로 원소를 적절한 위치에 삽입하고 나머지 원소는 하나씩 오른쪽으로 이동시킴
- 특징
  - 레코드를 계속 이동시켜야 하므로 레코드의 크기가 큰 경우에 불리
  - 거의 정렬이 된 화일인 경우 유리
  - 안정적인 제자리 정렬
  - 더미(dummy) 키가 필요함

삽입 과정 (1)



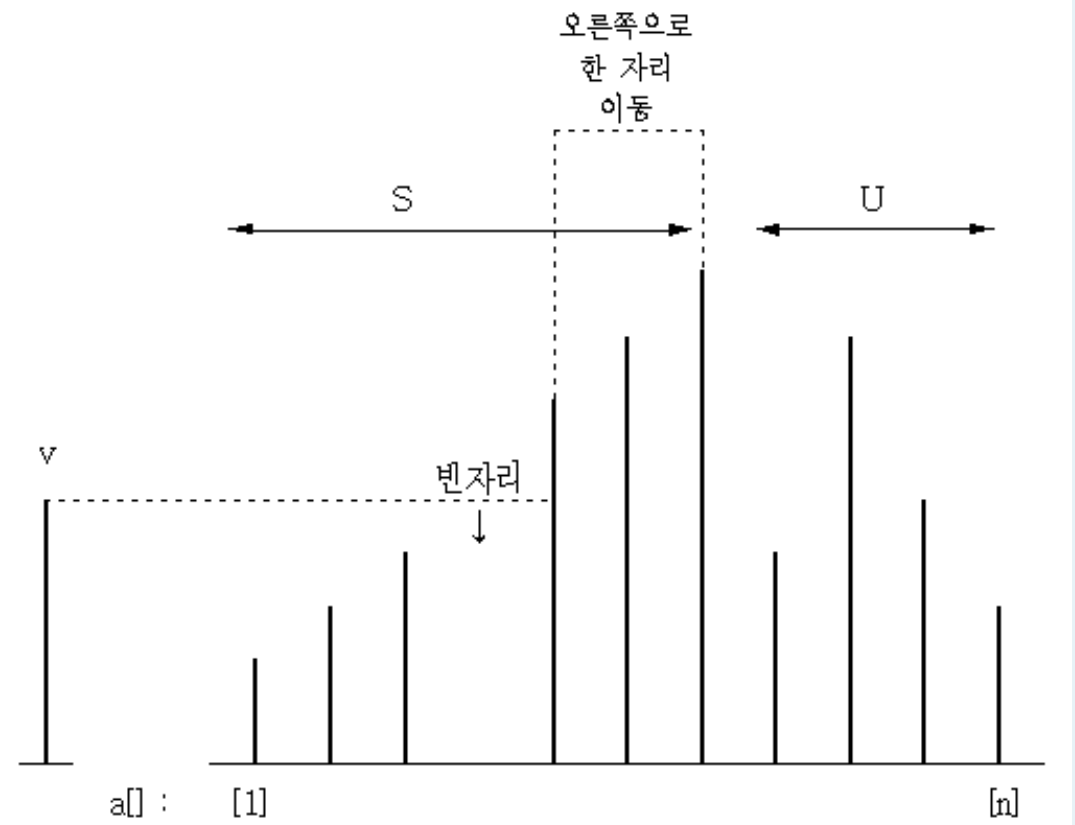
(a) 초기 단계

삽입 과정 (2)



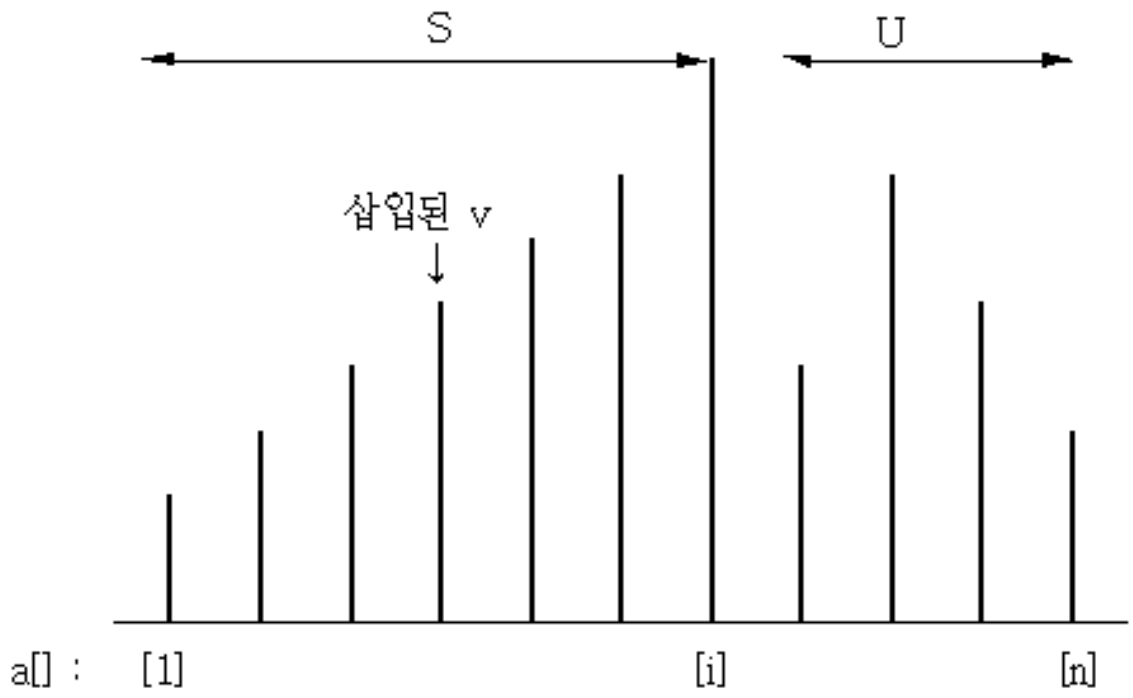
(b) a[i]를 제거하여 빈자리를 만듦(v ← a[i])

삽입 과정 (3)



(c) v보다 큰 원소는 오른쪽으로 한 자리씩 이동

삽입 과정 (4)

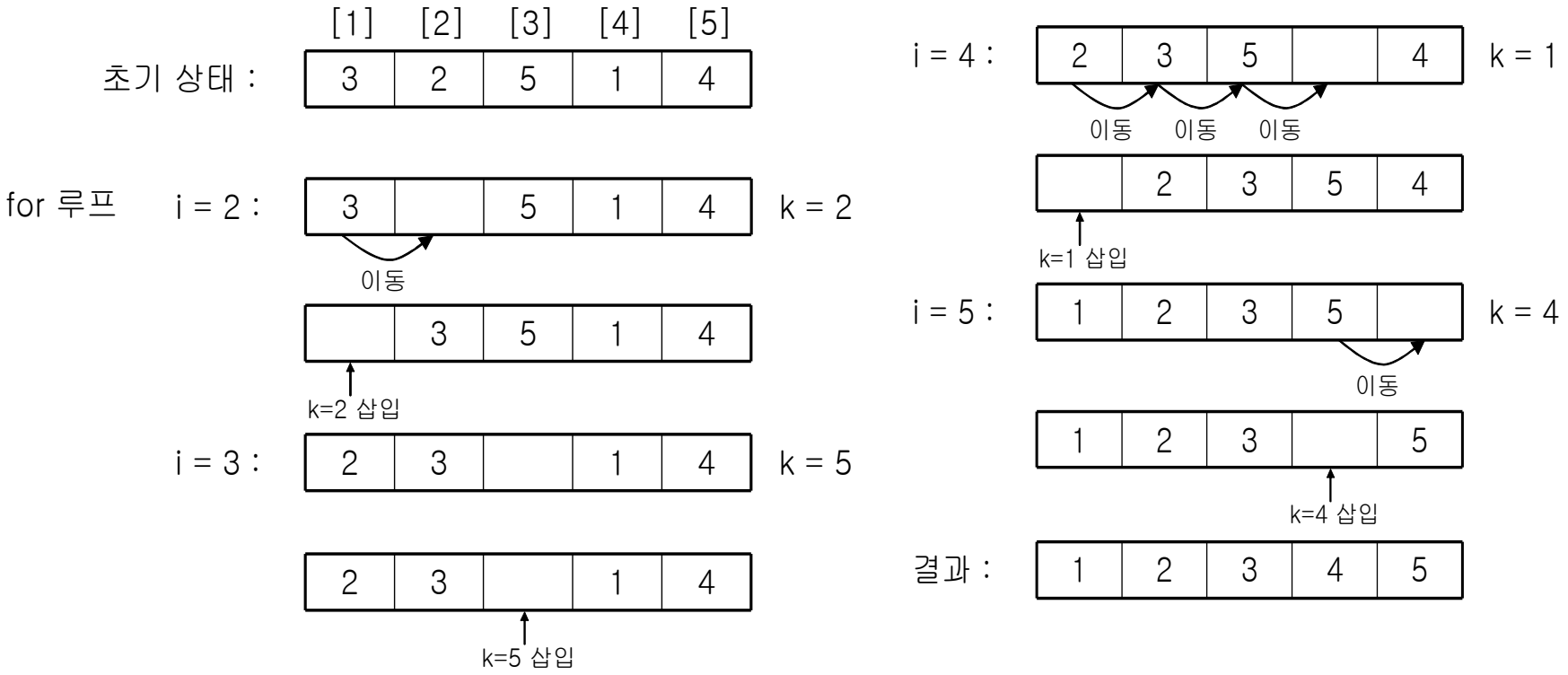


(d) v를 빈자리에 삽입

그림 3.3 삽입 정렬에서 원소 v(=a[i])의 삽입 과정

수행 과정

- a[] = (3, 2, 5, 1, 4)


[1] [2] [3] [4] [5]
초기 상태 :
3 2 5 1 4
for 루프
i = 2 :
3 5 1 4
k = 2
이동
3 5 1 4
k=2 삽입
i = 3 :
2 3 1 4
k = 5
2 3 1 4
k=5 삽입
i = 4 :
2 3 5 4
k = 1
이동 이동 이동
2 3 5 4
k=1 삽입
i = 5 :
1 2 3 5
k = 4
이동
1 2 3 5
k=4 삽입
결과 :
1 2 3 4 5

성능 특성

- 시간 복잡도 $O(N^2)$
- 거의 정렬된 화일의 경우 효율적임

쉘 정렬 (shellsort)

- 삽입 정렬을 간단하게 변형
- 멀리 떨어진 원소끼리 교환이 가능하게 하여 정렬 속도를 향상시킴
- *h*-정렬 화일 : 모든 *h*번째 원소를 정렬한 화일
- 인덱스 간격 순차의 예 : ..., 1093, 364, 121, 40, 13, 4, 1
- 특징
  - 쉘 정렬의 성능은 인덱스 간격 순차에 따라 달라짐
  - 최선의 경우 시간 복잡도는 $O(N \log N)$, 평균적인 경우는 $O(N^{4/3})$, 최악의 경우는 $O(N^{3/2})$
  - 제자리 정렬이지만 불안정적

수행 과정(1)

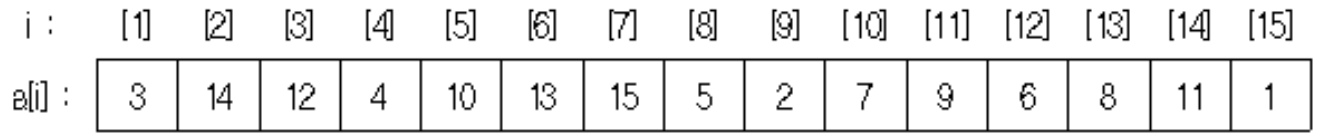


그림 3.5 배열 a[1 : 15]

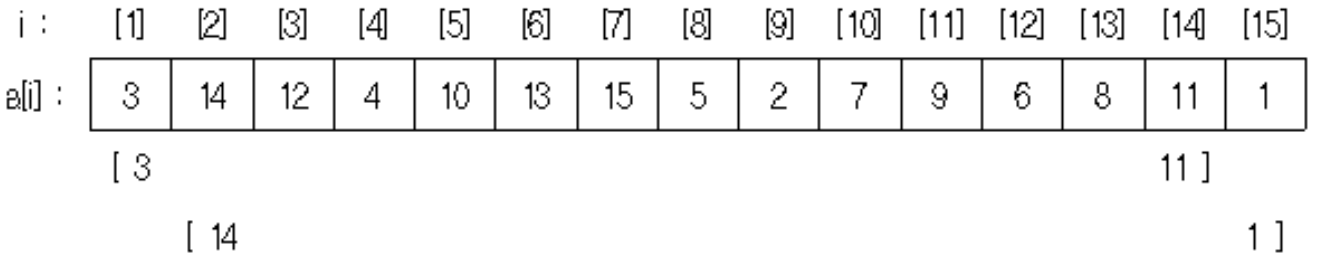


그림 3.6 2개의 h-13 서브리스트

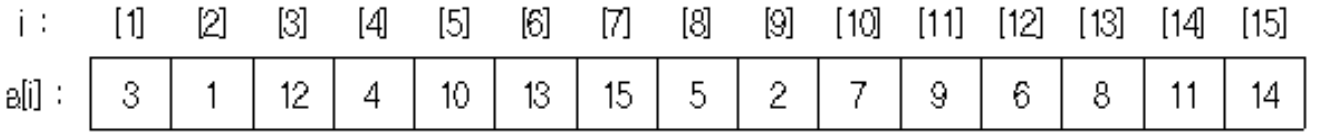
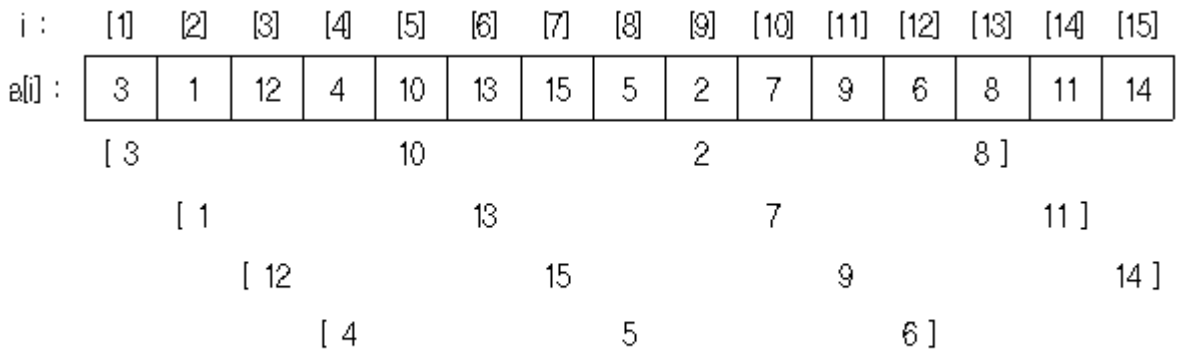


그림 3.7 정렬된 h-13 리스트

수행 과정(2)



[3 10 2 8] → [2 3 8 10]
[1 13 7 11] → [1 7 11 13]
[12 15 9 14] → [9 12 14 15]
[4 5 6] → [4 5 6]

그림 3.8 4개의 h-4 서브리스트

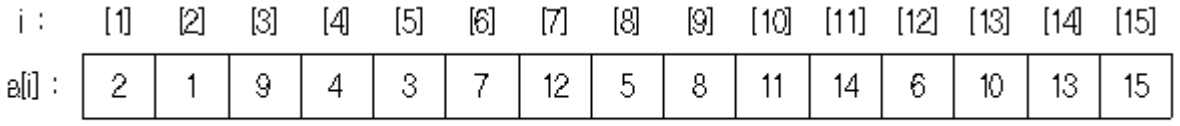


그림 3.9 정렬된 h-4 리스트

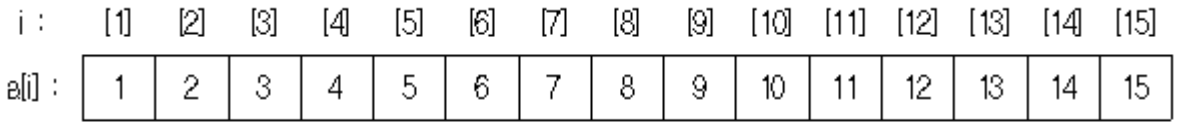


그림 3.10 최종 정렬 결과

퀵 정렬 (quicksort)

- 1960년 영국의 컴퓨터과학자 Tony Hoare가 개발
- 분할 정복(divide and conquer) 기법을 사용한 정렬 방법의 하나
- 현재 가장 광범위하게 쓰이는 정렬 알고리즘
- 지금까지 퀵 정렬의 성능을 개선하려는 시도가 있었지만 큰 성과를 거두지 못함
- 특징
  - 평균적으로 아주 좋은 성능을 가짐
  - 약간의 작은 스택만 있으면 별도의 메모리를 요구하지 않음
  - 불안정적인 제자리 알고리즘
  - N 개의 원소를 정렬하는데 평균적으로 $N \log N$ 의 연산속도를 가짐

퀵 정렬 – 기본 알고리즘

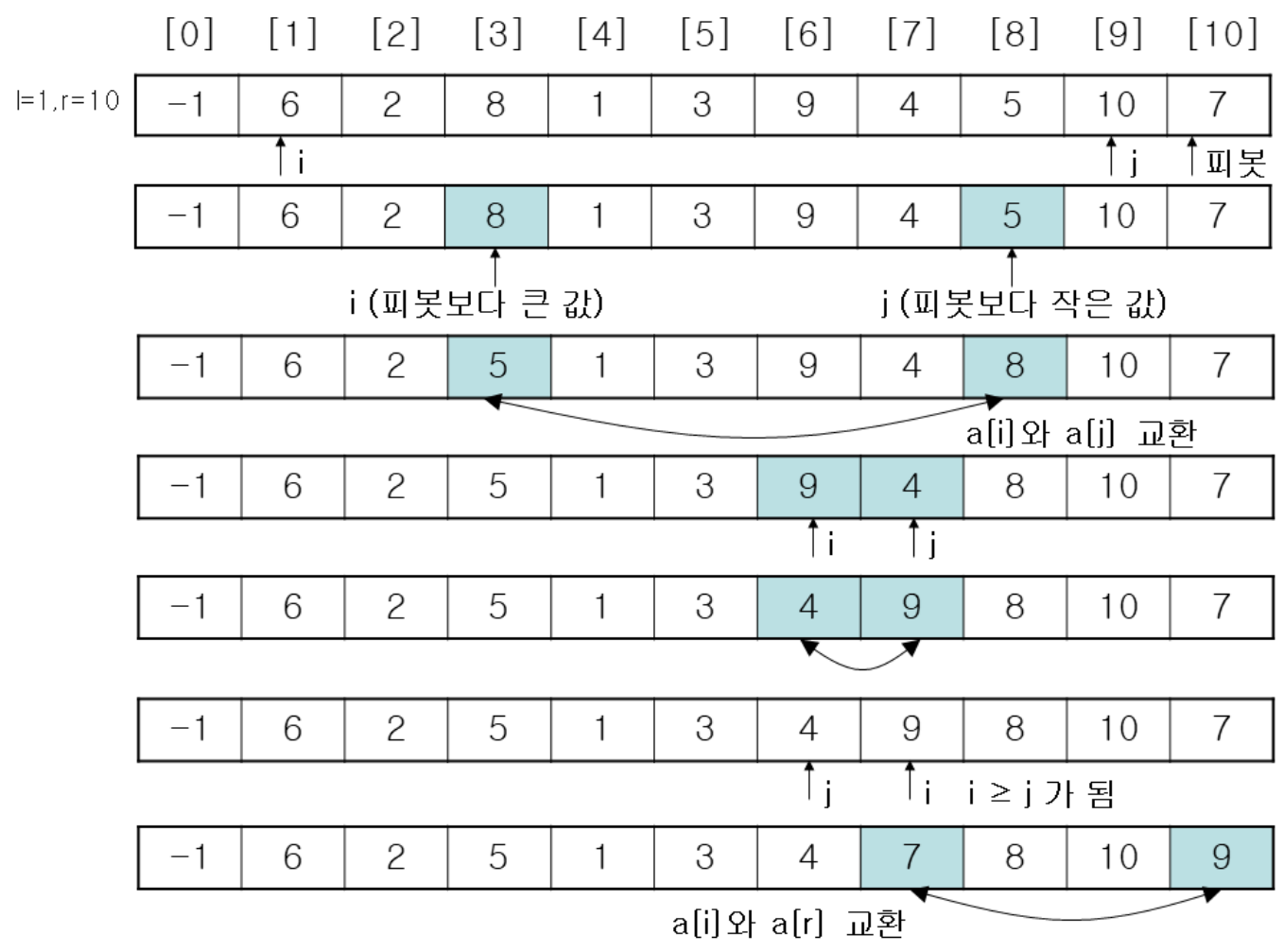
1. 배열 a[l : r] a[r]을 피봇(pivot)으로 선정
2. 피봇을 기준으로 a[]의 원소들을 두 개의 파티션(partition)으로 분할
   - 분할 후 피봇은 a[i]에 들어가게 되는데, 이곳은 정렬 후 피봇이 들어가는 정확한 위치가 됨
   - 왼쪽 파티션에 있는 원소 a[l], ..., a[i-1] 중 피봇보다 큰 원소는 없음
   - 오른쪽 파티션에 있는 원소 a[i+1], ..., a[r] 중 피봇보다 작은 원소는 없음

알고리즘

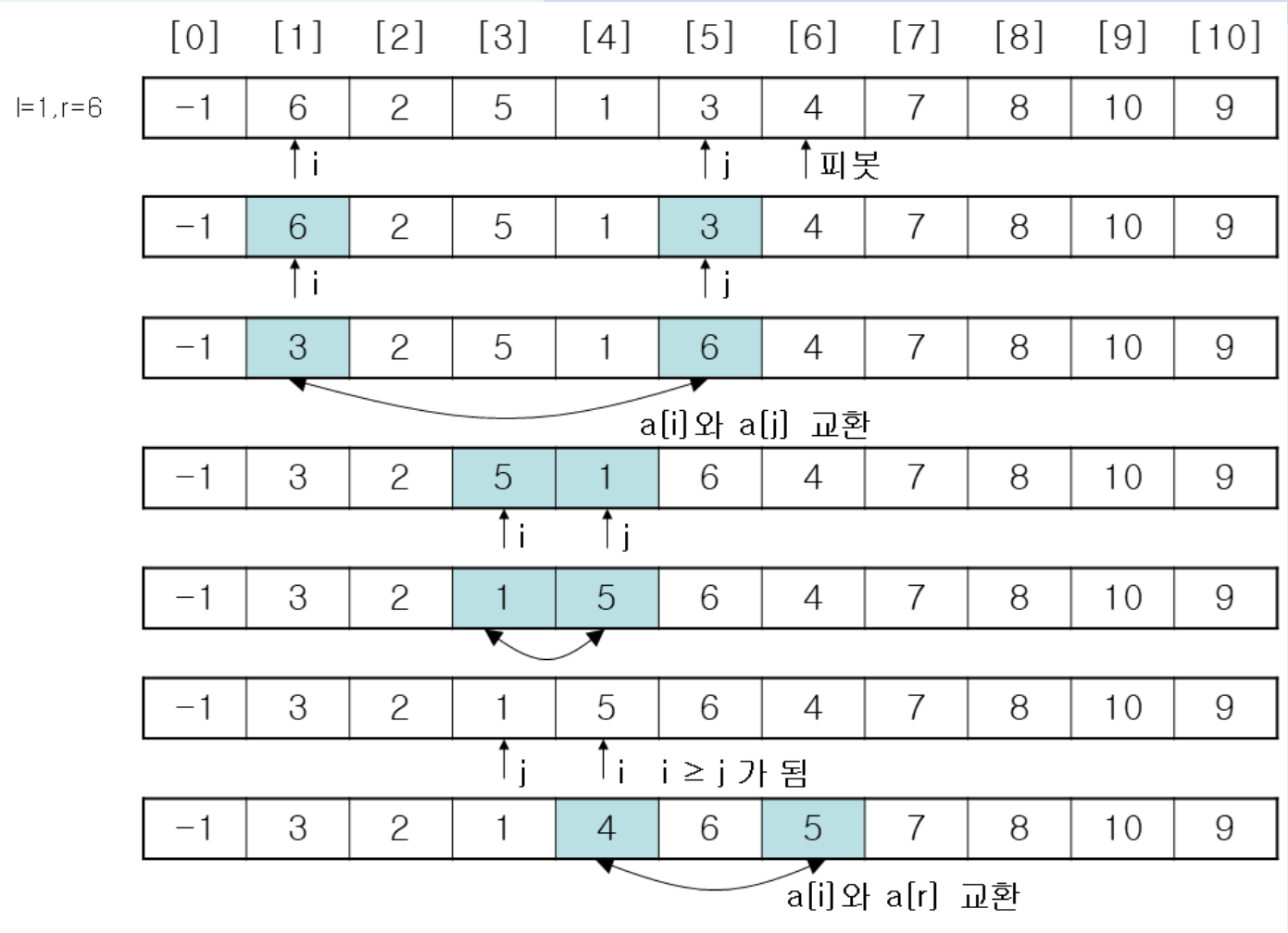
◆ 퀵 정렬 알고리즘

```
quickSort(a[], l, r)
  // 배열 a[]의 부분 배열 a[l : r]을 오름차순으로 정렬
  if (r > l) then {
    i ← partition(a[], l, r);   // i는 파티션이 끝난 뒤에 사용된 피봇의 인덱스
    quickSort(a[], l, i-1);
    quickSort(a[], i+1, r);
  }
end quickSort()
```

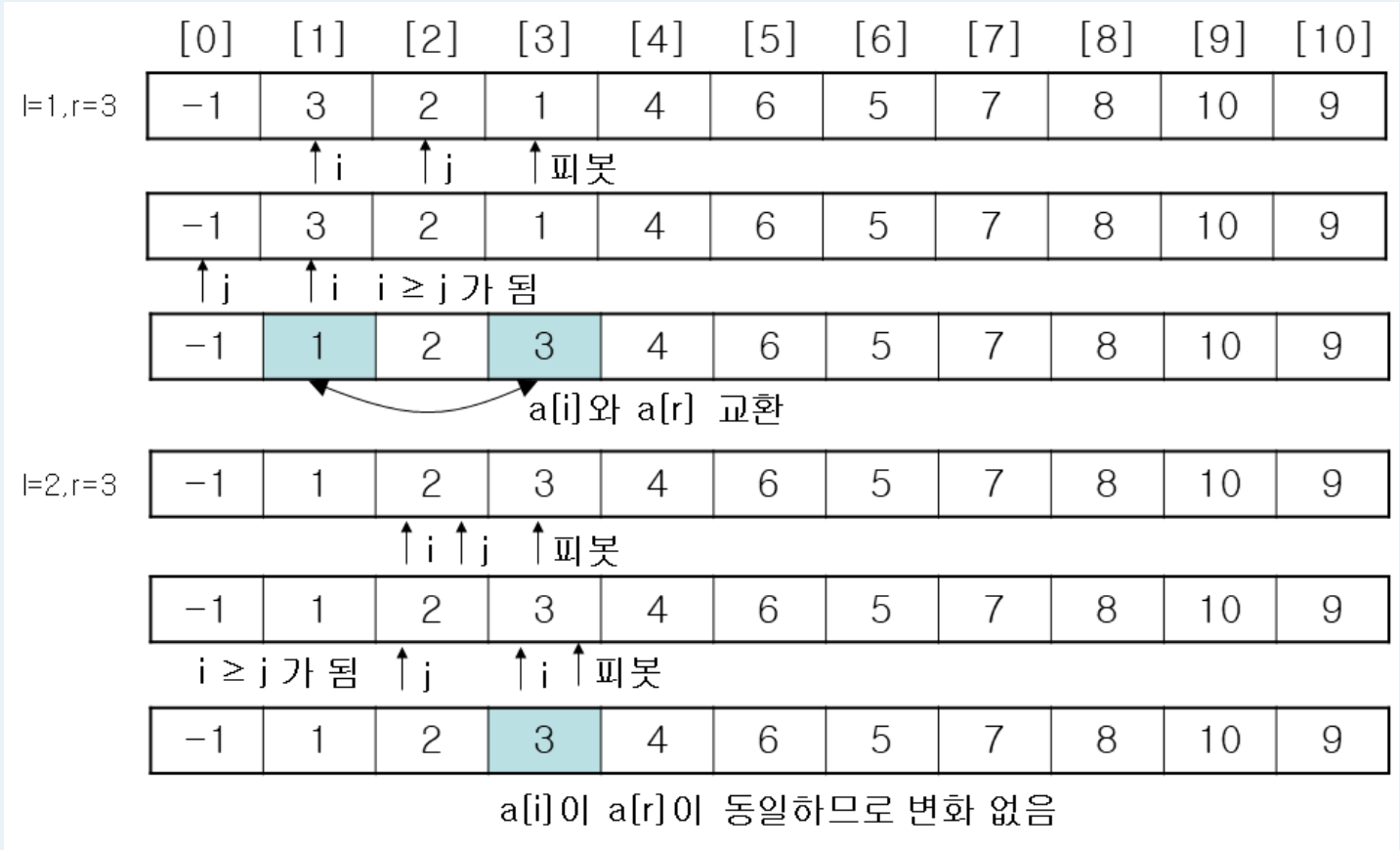
수행과정(1)


[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
l=1,r=10
-1 6 2 8 1 3 9 4 5 10 7
i
j
피봇
-1 6 2 8 1 3 9 4 5 10 7
i (피봇보다 큰 값)
j (피봇보다 작은 값)
-1 6 2 5 1 3 9 4 8 10 7
a[i]와 a[j] 교환
-1 6 2 5 1 3 9 4 8 10 7
i
j
-1 6 2 5 1 3 4 9 8 10 7
-1 6 2 5 1 3 4 9 8 10 7
j
i
i ≥ j 가 됨
-1 6 2 5 1 3 4 7 8 10 9
a[i]와 a[r] 교환

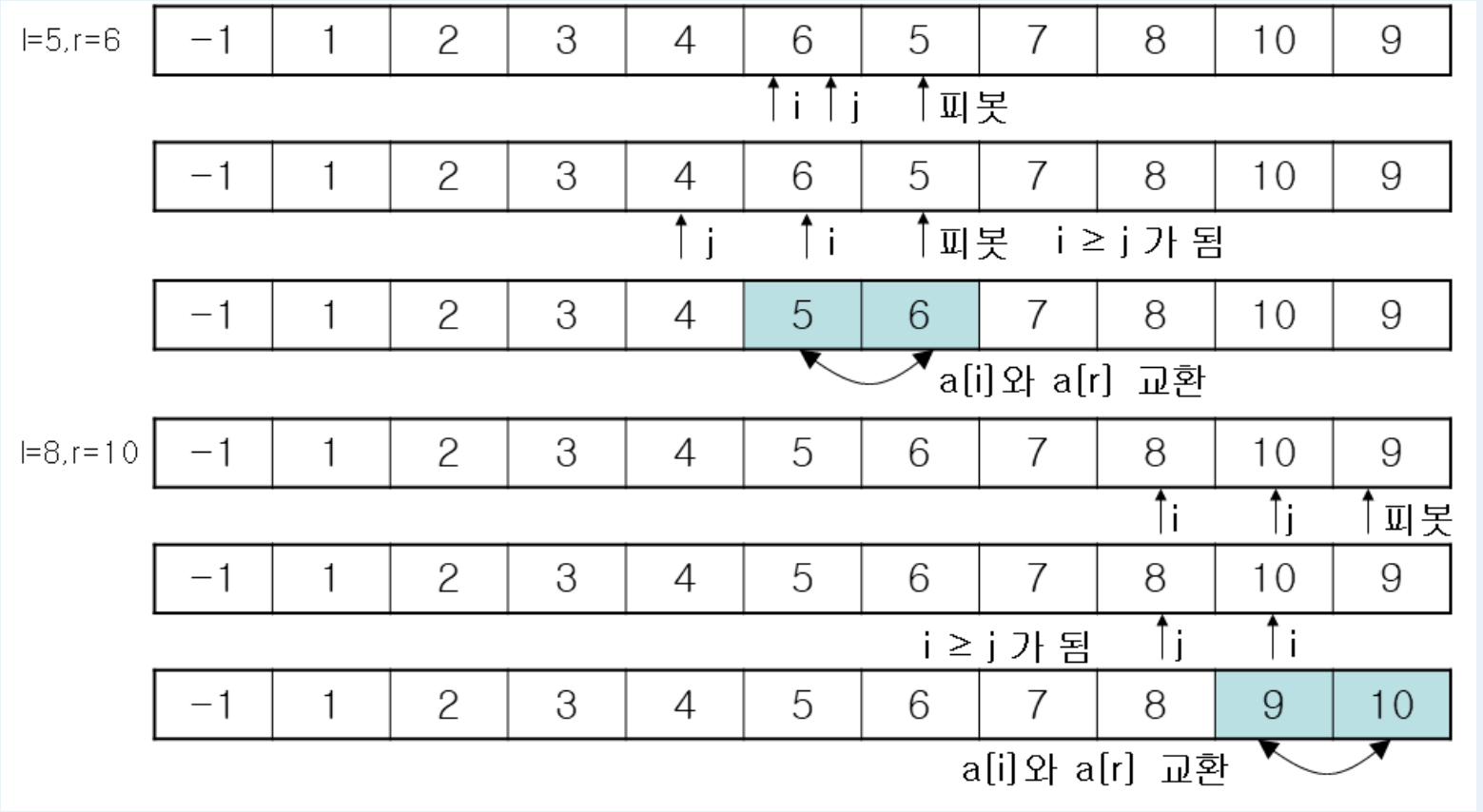
수행과정(2)


[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
l=1,r=6
-1 6 2 5 1 3 4 7 8 10 9
i
j
피봇
-1 6 2 5 1 3 4 7 8 10 9
i
j
-1 3 2 5 1 6 4 7 8 10 9
a[i]와 a[j] 교환
-1 3 2 5 1 6 4 7 8 10 9
i
j
-1 3 2 1 5 6 4 7 8 10 9
-1 3 2 1 5 6 4 7 8 10 9
j
i
i ≥ j 가 됨
-1 3 2 1 4 6 5 7 8 10 9
a[i]와 a[r] 교환

수행과정(3)


[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
l=1,r=3
-1 3 2 1 4 6 5 7 8 10 9
i j 피봇
-1 3 2 1 4 6 5 7 8 10 9
j i i ≥ j 가 됨
-1 1 2 3 4 6 5 7 8 10 9
a[i]와 a[r] 교환
l=2,r=3
-1 1 2 3 4 6 5 7 8 10 9
i j 피봇
-1 1 2 3 4 6 5 7 8 10 9
i ≥ j 가 됨 j i 피봇
-1 1 2 3 4 6 5 7 8 10 9
a[i]이 a[r]이 동일하므로 변화 없음

수행과정(4)


l=5,r=6
-1 1 2 3 4 6 5 7 8 10 9
i j 피봇
-1 1 2 3 4 6 5 7 8 10 9
j i 피봇 i ≥ j 가 됨
-1 1 2 3 4 5 6 7 8 10 9
a[i]와 a[r] 교환
l=8,r=10
-1 1 2 3 4 5 6 7 8 10 9
i j 피봇
-1 1 2 3 4 5 6 7 8 10 9
i ≥ j 가 됨 j i
-1 1 2 3 4 5 6 7 8 9 10
a[i]와 a[r] 교환

성능 특성

- 최선의 경우
  - $C_N = 2C_{N/2} + N$
  - $C_N \approx N \log N$
- 평균
  - $C_N \approx 2N \ln N$
- 평균 비교 횟수는 최선의 경우에 비해 약 38% 정도 많아지므로 큰 차이가 나지 않는다고 할 수 있음
  - $2N \ln N \approx 1.38\ N \log N$

퀵 정렬의 성능 향상 방법

- 작은 부분화일의 경우 삽입 정렬 사용
- 중간값 분할(median-of-three partitioning)

작은 부분화일

- 부분화일의 크기가 일정 크기 이하로 작아지면 삽입 정렬 수행
- “if r > l:”
  - “if r - l <= M: insertionSort(a, l, r)”
- M : 5 ~ 25
- 많은 응용에서 약 20% 정도의 시간 절감 효과가 있음

중간 값 분할

- 분할 원소를 선택할 때 왼쪽, 가운데, 오른쪽 원소 중 값이 중간인 원소를 선택
- 왼쪽, 가운데, 오른쪽 원소를 정렬한 후 가장 작은 값을 a[l], 가장 큰 값을 a[r], 중간 값을 a[r-1]에 넣고, a[l+1], …, a[r-2]에 대해 분할 알고리즘을 수행
- 장점
  - 최악의 경우가 발생하는 확률을 낮추어 줌
  - 경계 키(sentinel key)를 사용할 필요가 없음
  - 전체 수행 시간을 약 5% 감소시킴

합병 정렬(mergesort)

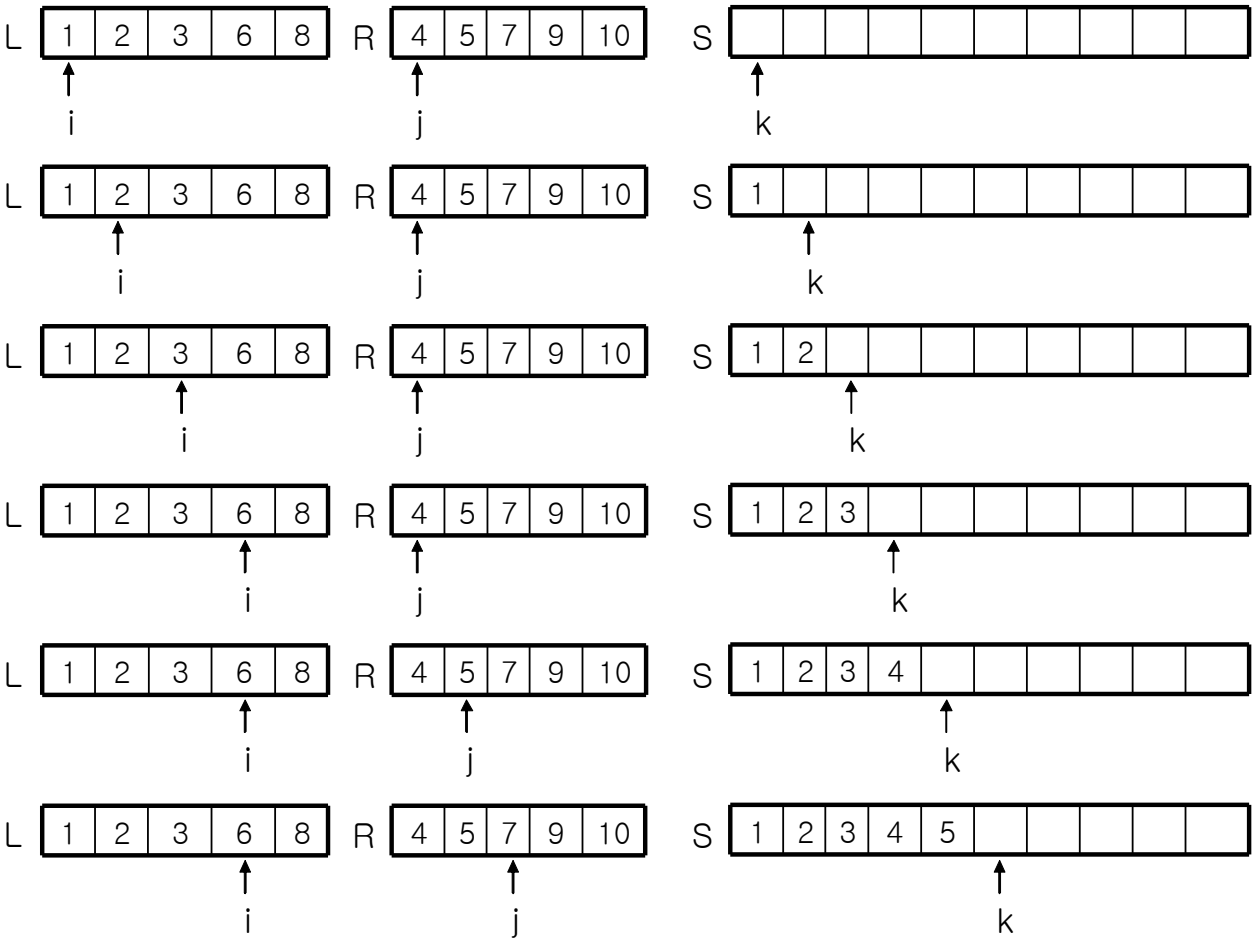
- 두 개의 정렬된 화일을 하나의 큰 정렬된 화일로 합병함
- 합병 정렬은 퀵 정렬과 마찬가지로 분할 정복 방식의 알고리즘임
- 두 부분배열의 크기가 동일하도록 분할함

알고리즘

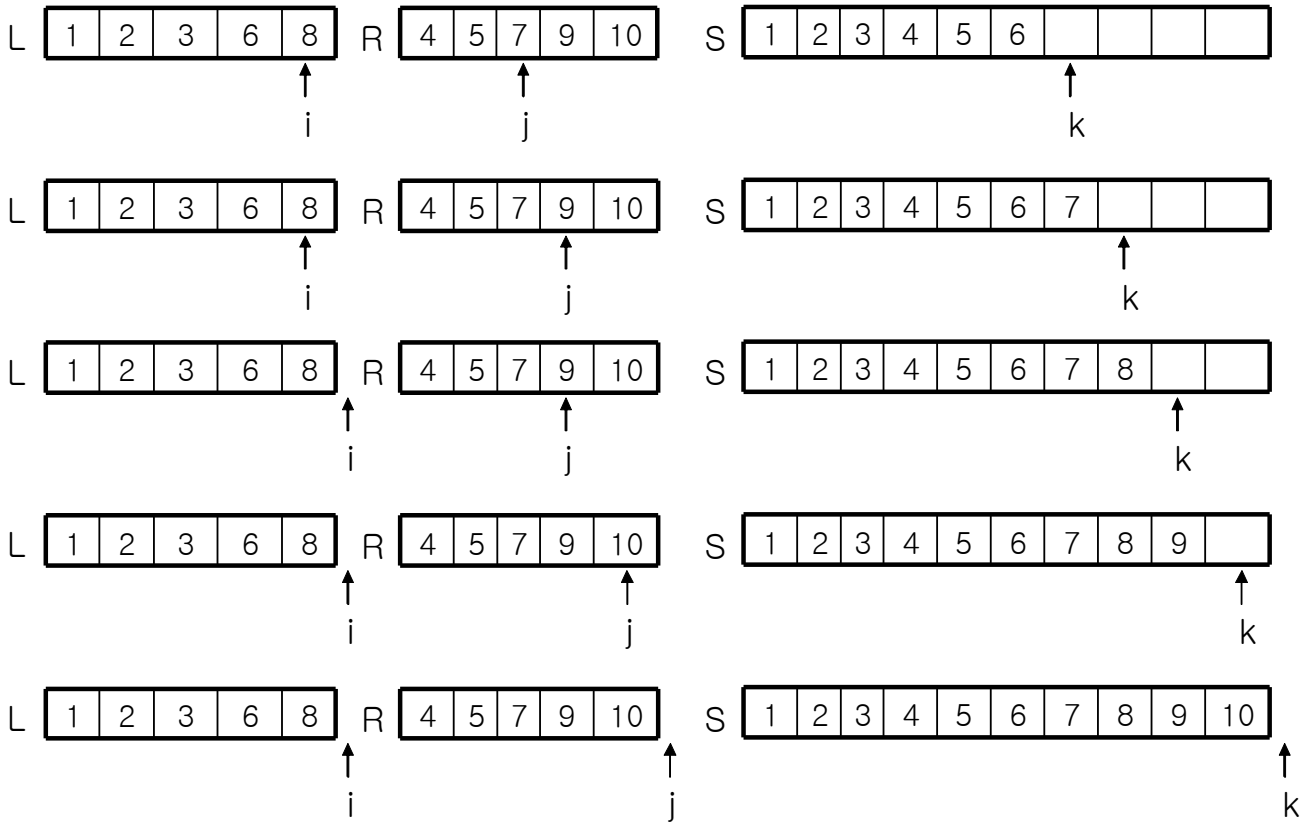
◆ 합병 정렬 알고리즘

```
mergeSort(a[], l, r)
  if (r > l) then {
    m ← (r+l)/2;
    mergeSort(a[], l, m);
    mergeSort(a[], m+1, r);
    merge(a[], l, m, r);
  }
end mergeSort()
```

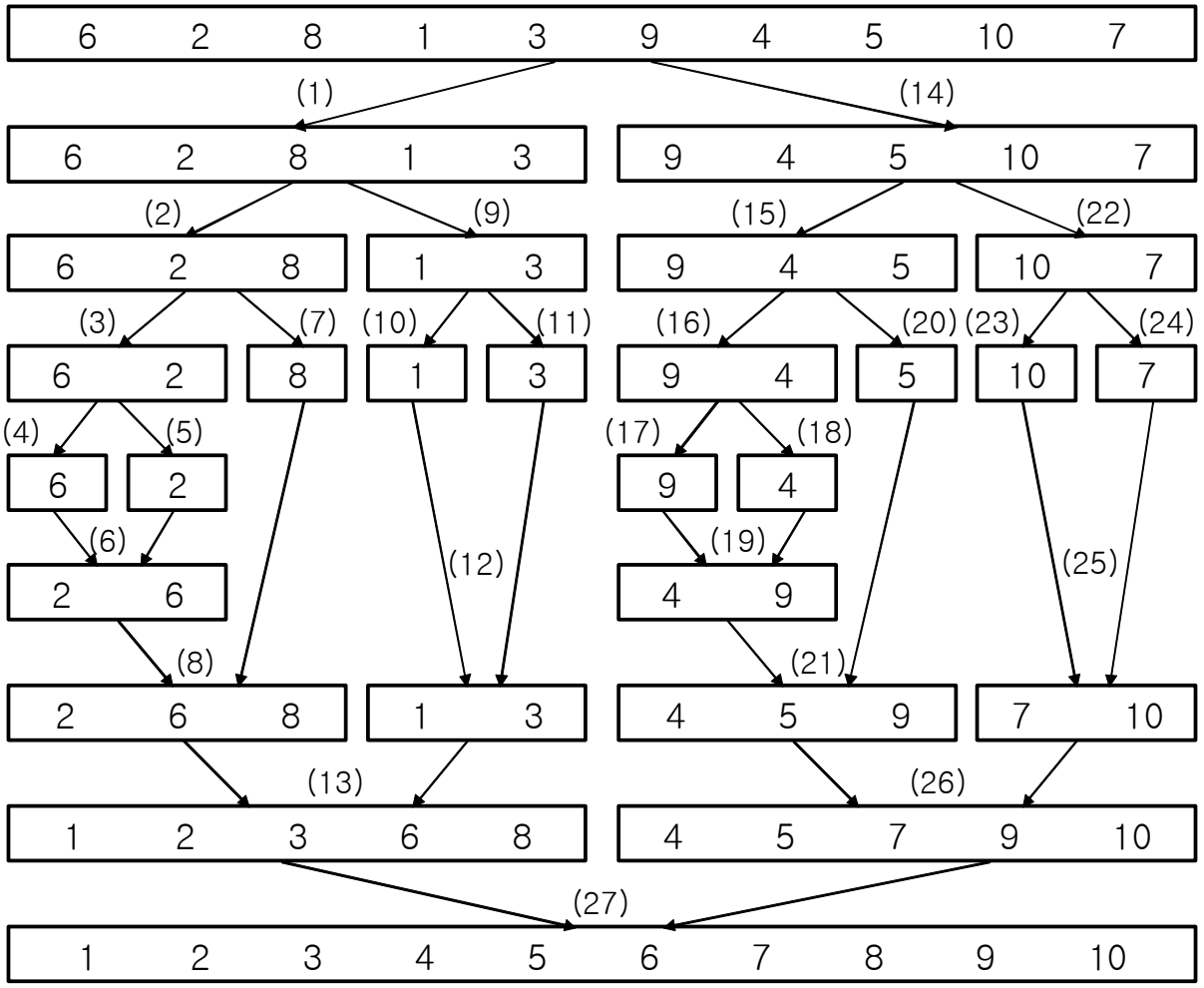
합병 알고리즘(1)


L 1 2 3 6 8
R 4 5 7 9 10
S
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1 2
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1 2 3
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1 2 3 4
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1 2 3 4 5
i
j
k

합병 알고리즘(2)


L 1 2 3 6 8
R 4 5 7 9 10
S 1 2 3 4 5 6
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1 2 3 4 5 6 7
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1 2 3 4 5 6 7 8
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1 2 3 4 5 6 7 8 9
i
j
k
L 1 2 3 6 8
R 4 5 7 9 10
S 1 2 3 4 5 6 7 8 9 10
i
j
k

수행 과정


6 2 8 1 3 9 4 5 10 7
(1)
(14)
6 2 8 1 3
9 4 5 10 7
(2)
(9)
(15)
(22)
6 2 8
1 3
9 4 5
10 7
(3)
(7)
(10)
(11)
(16)
(20)
(23)
(24)
6 2
8
1
3
9 4
5
10
7
(4)
(5)
(17)
(18)
6
2
9
4
(6)
(19)
(12)
(25)
2 6
4 9
(8)
(21)
2 6 8
1 3
4 5 9
7 10
(13)
(26)
1 2 3 6 8
4 5 7 9 10
(27)
1 2 3 4 5 6 7 8 9 10

성능 특성

- 최악의 경우에도 N 개의 원소를 가진 화일을 정렬할 때의 시간 복잡도는 $O(N \log N)$
- 순차적 방식에 의해 데이터를 접근함
  - 연결 리스트와 같이 순차 접근이 유일한 접근 방법일 경우 사용 가능
- 입력 배열에 민감하지 않음
- 안정적이지만 제자리 정렬이 아님
  - N 에 비례하는 추가 기억장소가 필요함

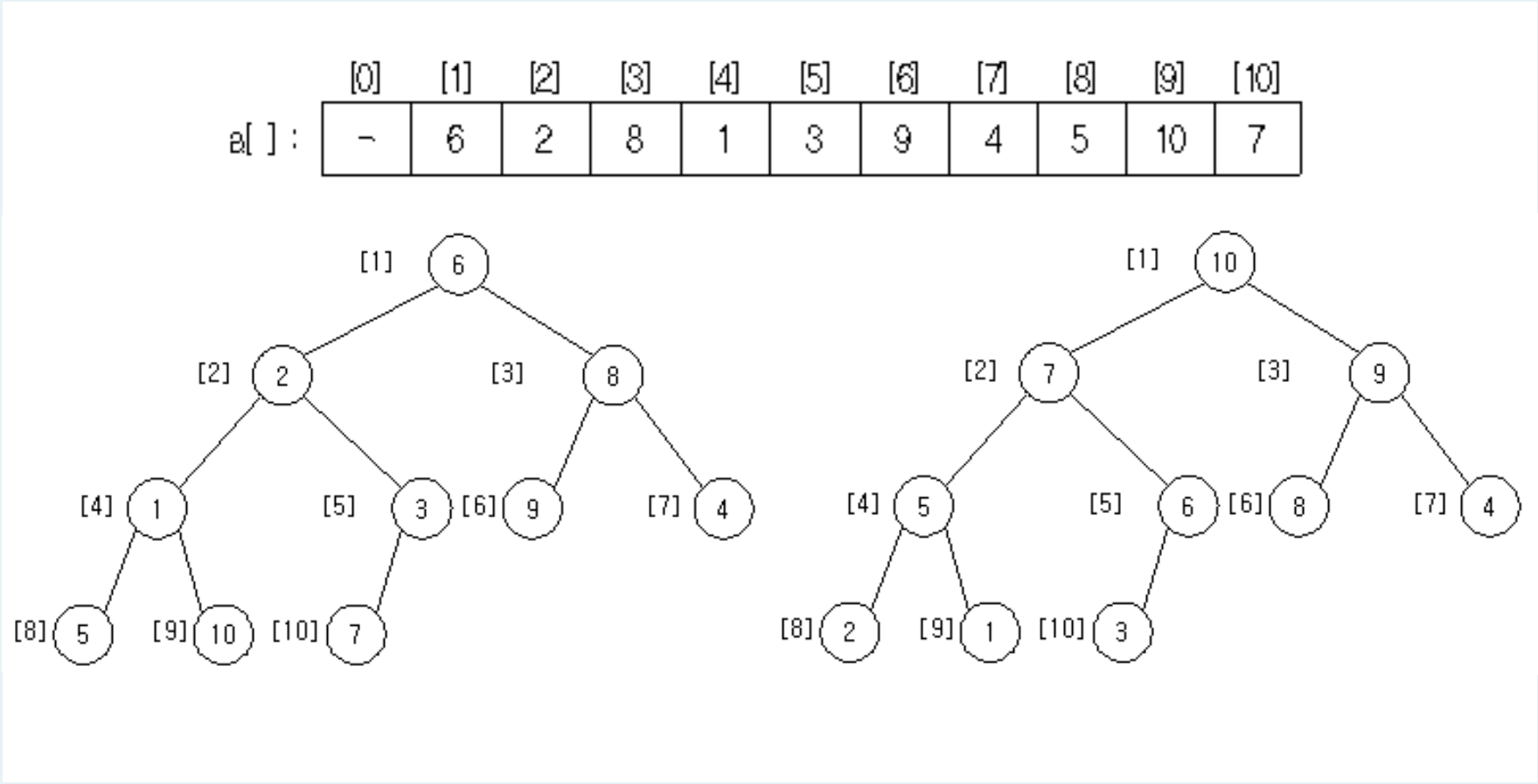
퀵 정렬과 비교

- 퀵 정렬 : 정복-분할(conquer-and-divide)
  - 순환 호출이 이루어지기 전에 대부분의 작업이 수행
  - 가장 큰 부분화일로부터 시작하여 가장 작은 부분화일에서 종료됨 → 스택이 필요함
  - 불안정적
- 합병 정렬 : 분할-정복(divide-and-conquer)
  - 처음에 화일을 두 부분으로 분할하고 나서, 각각의 부분을 개별적으로 정복함
  - 가장 작은 부분화일로부터 시작하여 가장 큰 부분화일에서 종료됨 → 스택이 필요 없음
  - 안정적

히프 정렬(heapsort)

- 히프(heap)를 이용해 정렬
  - 히프 : 우선순위 큐의 일종
  - 정렬할 원소를 모두 공백 히프에 하나씩 삽입
  - 한 원소씩 삭제 →제일 큰 원소가 삭제됨
  - 이 원소를 리스트의 뒤에서부터 차례로 삽입
  - 오름차순으로 정렬된 리스트를 생성

히프 구조


[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
a[ ] : - 6 2 8 1 3 9 4 5 10 7
[1] 6
[2] 2
[3] 8
[4] 1
[5] 3
[6] 9
[7] 4
[8] 5
[9] 10
[10] 7
[1] 10
[2] 7
[3] 9
[4] 5
[5] 6
[6] 8
[7] 4
[8] 2
[9] 1
[10] 3

알고리즘

◆ 히프 정렬 알고리즘

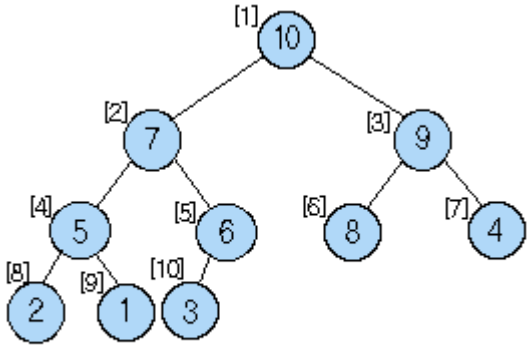
```
heapSort(a[], n)
  for (i ← n/2; i ≥ 1; i ← i-1) do        // 배열 a[]를 히프로 변환
    heapify(a, i, n);                     // i는 내부 노드
  for (i ← n-1; i ≥ 1; i ← i-1) do {      // 배열 a[]를 오름차순으로 정렬
    a[1]과 a[i+1]을 교환;    // a[1]은 제일 큰 원소
    heapify(a, 1, i);
  }
end heapSort()
```

수행 과정(1)

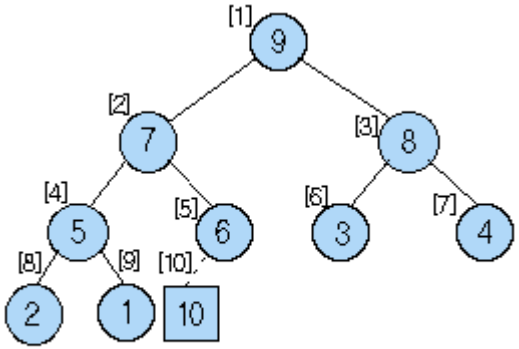
a[ ] :	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
초기 :	6	2	8	1	3	9	4	5	10	7
히프 (크기 10) 로 변환 :	10	7	9	5	6	8	4	2	1	3
(두 번째 for 루프) i = 9 :	9	7	8	5	6	3	4	2	1	10*
(i = 히프 크기) i = 8 :	8	7	4	5	6	3	1	2	9*	
i = 7 :	7	6	4	5	2	3	1	8*		
i = 6 :	6	5	4	1	2	3	7*			
i = 5 :	5	3	4	1	2	6*				
i = 4 :	4	3	2	1	5*					
i = 3 :	3	1	2	4*						
i = 2 :	2	1	3*							
i = 1 :	1*	2*								

점선은 정렬이 완료된 원소 값

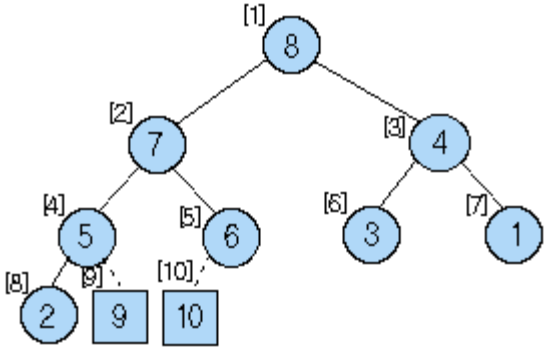
수행 과정(2)



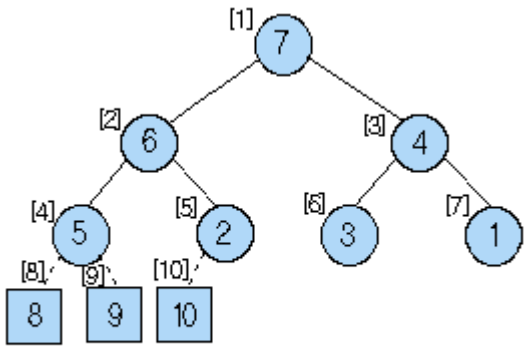
(a) 히프 크기=10



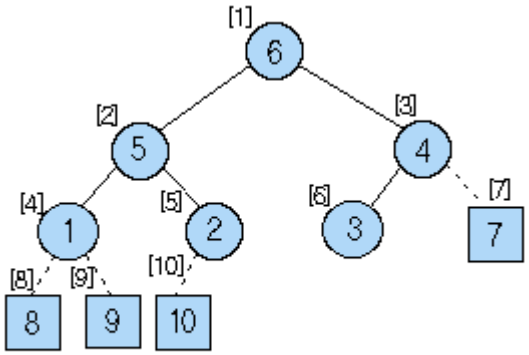
(b) 히프 크기=9



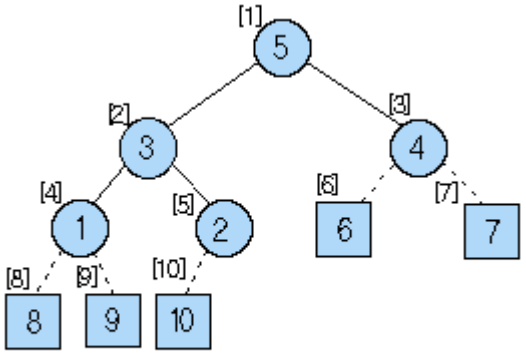
(c) 히프 크기=8



(d) 히프 크기=7

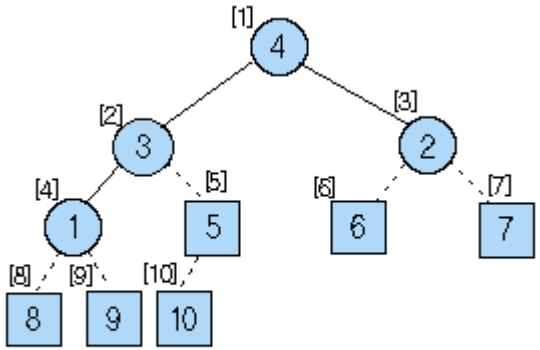


(e) 히프 크기=6

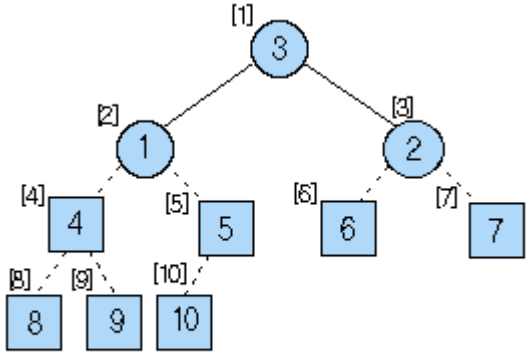


(f) 히프 크기=5

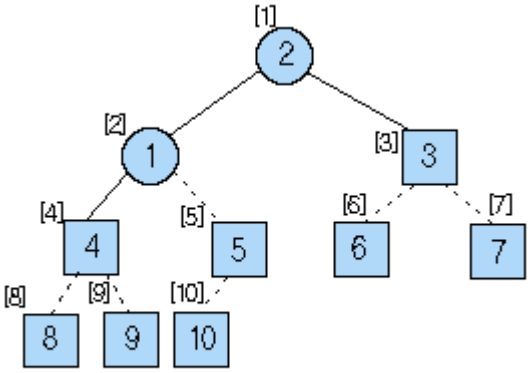
수행 과정(3)



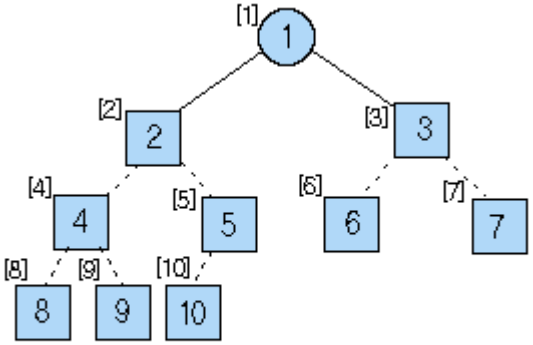
(g) 히프 크기=4



(h) 히프 크기=3



(i) 히프 크기=2



(j) 히프 크기=1;정렬 종료

성능 특성

- 제자리 정렬이지만 불안정적
- N 개의 원소를 정렬할 때 최악의 경우 시간 복잡도는 $O(N \log N)$
- 입력 배열의 순서에 민감하지 않음
- 내부 루프가 퀵 정렬보다 약간 길어서 평균적으로 퀵 정렬보다 2배 정도 느림

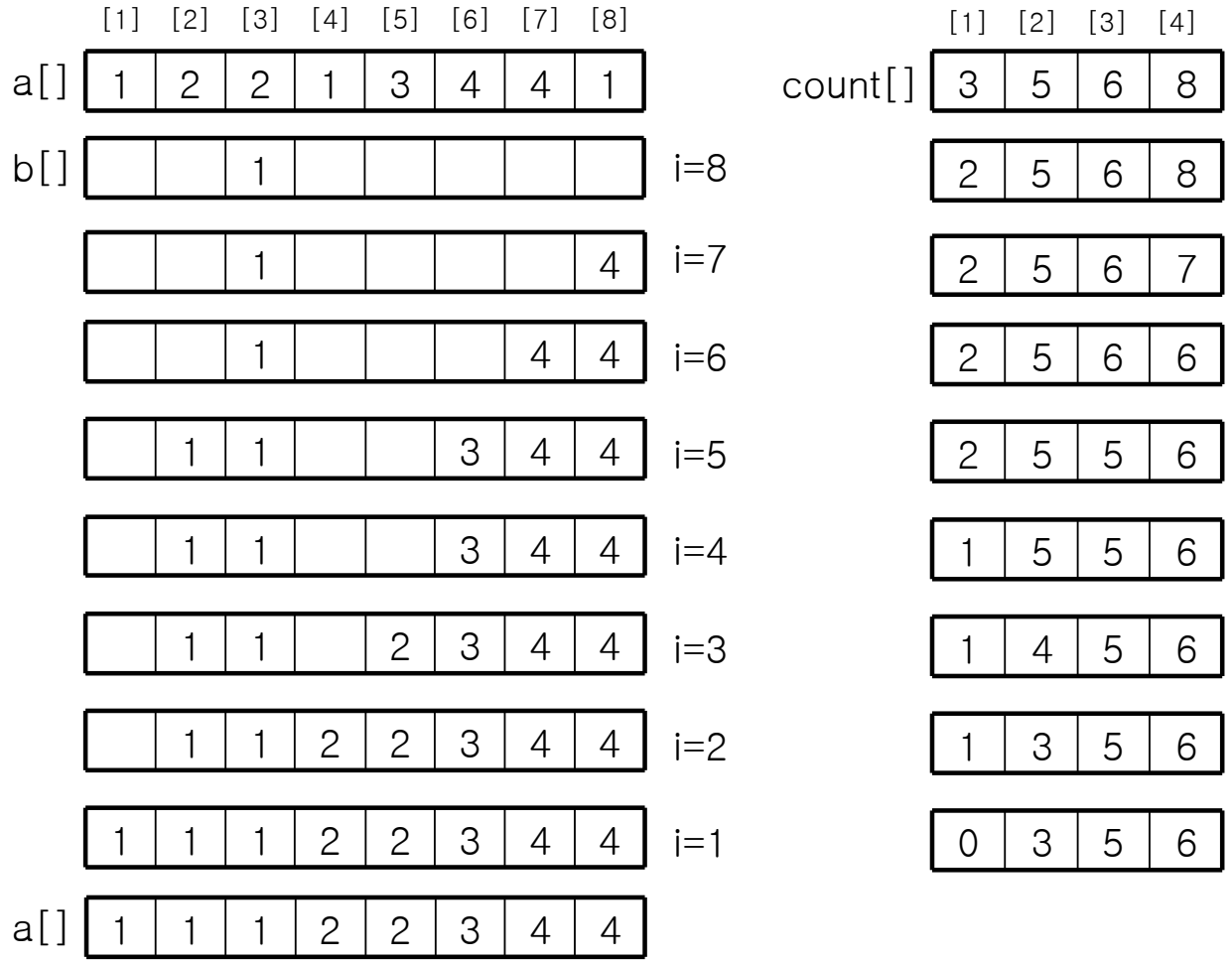
분포에 의한 정렬

- 비교 기반 정렬 알고리즘
  - 최악의 경우 비교 횟수가 $O(N \log N)$임이 증명됨
  - 따라서 최악의 경우 시간 복잡도가 $O(N \log N)$ 미만인 알고리즘은 구할 수 없음
- 키의 분포를 이용한 정렬 알고리즘
  - 최악 또는 평균 실행시간이 $O(N)$인 알고리즘
  - 계수 정렬, 기수 정렬, 버킷 정렬

계수 정렬(Counting Sort)

- 적용 범위
  - 입력 키가 어떤 범위에 있을 때 적용 가능
  - 예를 들어, 입력 키가 1부터 k 사이의 작은 정수 범위에 있다는 것을 미리 알고 있을 때에만 적용 가능

수행 과정


[1] [2] [3] [4] [5] [6] [7] [8]
a[] 1 2 2 1 3 4 4 1
b[] 1 i=8
1 4 i=7
1 4 4 i=6
1 1 3 4 4 i=5
1 1 3 4 4 i=4
1 1 2 3 4 4 i=3
1 1 2 2 3 4 4 i=2
1 1 1 2 2 3 4 4 i=1
a[] 1 1 1 2 2 3 4 4
[1] [2] [3] [4]
count[] 3 5 6 8
2 5 6 8
2 5 6 7
2 5 6 6
2 5 5 6
1 5 5 6
1 4 5 6
1 3 5 6
0 3 5 6

성능 특성

- 시간 복잡도는 $O(N)$
  - 중첩된 for 루프가 없음
- 안정적
- N에 비례하는 추가 기억장소가 필요하기 때문에 제자리 정렬은 아님
- 비교 기반 정렬 알고리즘에 비해 빠름

기수 정렬(radixsort)

- 전체 키를 여러 자리로 나누어 각 자리마다 계수 정렬과 같은 안정적인 정렬 알고리즘을 적용하여 정렬하는 방법
- d 자리수 숫자들에 대하여 계수 정렬로 정렬
  - 각 자리수마다 $O(N)$ 시간이 걸리므로 전체로는 $O(dN)$ 시간이 걸리는데, d를 상수로 취급할 수 있다면 $O(N)$ 시간이 걸리게 됨
- 전체 데이터 개수만큼의 기억 장소와 진법 크기 만큼의 기억 장소가 추가로 필요함

성능 특성

- 키가 *m*자리 숫자로 되어 있는 경우 *m*번의 패스를 반복 수행
- *N* 개의 원소에 대해 이 연산의 시간 복잡도는 $O(N)$

수행 과정

A = (35, 81, 12, 67, 93, 46, 23, 26)

```
Q[0] :      [    ]
Q[1] :      [ 81 ]
Q[2] :      [ 12 ]
Q[3] :      [ 93   23 ]
Q[4] :      [    ]
Q[5] :      [ 35 ]
Q[6] :      [ 46   26 ]
Q[7] :      [ 67 ]
Q[8] :      [    ]
Q[9] :      [    ]
```

결과 리스트 : (81 12 93 23 35 46 26 67)

첫 번째 자리 수를 기초로 정렬

```
Q[0] :      [    ]
Q[1] :      [ 12 ]
Q[2] :      [ 23   26 ]
Q[3] :      [ 35 ]
Q[4] :      [ 46 ]
Q[5] :      [    ]
Q[6] :      [ 67 ]
Q[7] :      [    ]
Q[8] :      [ 81 ]
Q[9] :      [ 93 ]
```

결과 리스트 : (12 23 26 35 46 67 81 93)

두 번째 자리 수를 기초로 정렬

외부 정렬(external sorting)

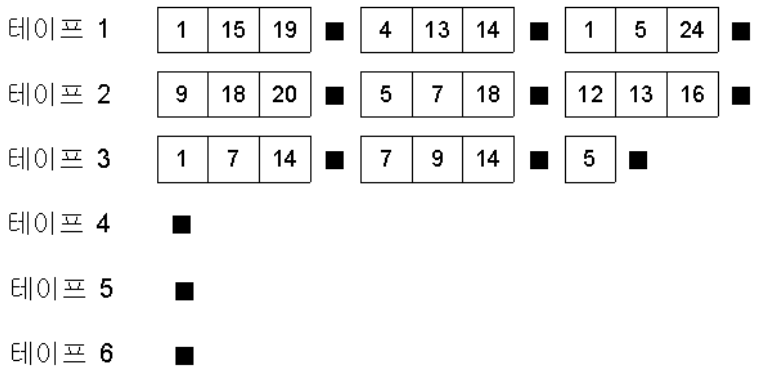
- 주기억 장치에 모두 적재해서 실행하기가 불가능한 매우 큰 데이터 화일을 정렬하는 기법
- 특징
  - 자료가 주기억 장치와 보조 기억 장치 사이를 오고가는데 드는 시간은 주기억 장치에 저장되어 있는 자료들을 서로 비교하는데 드는 시간에 비해 상대가 안될 정도로 오래 걸림
  - 외부 저장장치의 종류에 따라 접근 방식이 매우 제한적일 수 있음
- 외부 정렬의 가장 큰 비용은 입출력(input-output) 비용이므로 입출력 횟수를 줄이는 것이 알고리즘의 핵심임

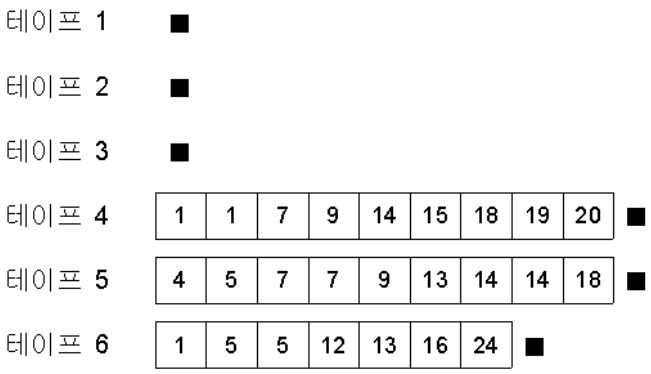
균형적 다방향 합병 정렬

- 균형적
  - 동일한 개수를 가지도록 블록들을 테이프에 분산
- 정렬 순서
  - 테이프에 저장되어 있는 화일을 주기억 장치에 옮겨올 수 있도록 작은 크기의 블록으로 나눔
  - 블록들을 한 개씩 주기억 장치에 읽어 들여서 내부 정렬을 수행함
  - 정렬된 작은 블록들을 동일한 개수를 가지도록 여러 개의 입력 테이프에 분산시켜 저장함
  - 정렬된 블록에서 원소들을 꺼내어 주기억 장치에서 합병한 뒤 이들을 다시 출력 테이프에 저장함
  - 한다. 합병이 계속되면 블록의 개수는 줄고, 각 블록의 크기는 커지게 됨
  - 합병 단계에서 하나의 블록만 남게 되면 정렬이 완료됨

수행 과정(1)

- L = (1, 19, 15, 18, 20, 9, 14, 7, 1, 14, 4, 13, 5, 18, 7, 9, 14, 7, 5, 24, 1, 13, 16, 12, 5)
- 주기억 장치는 3개의 레코드를 저장





대치 선택(1)

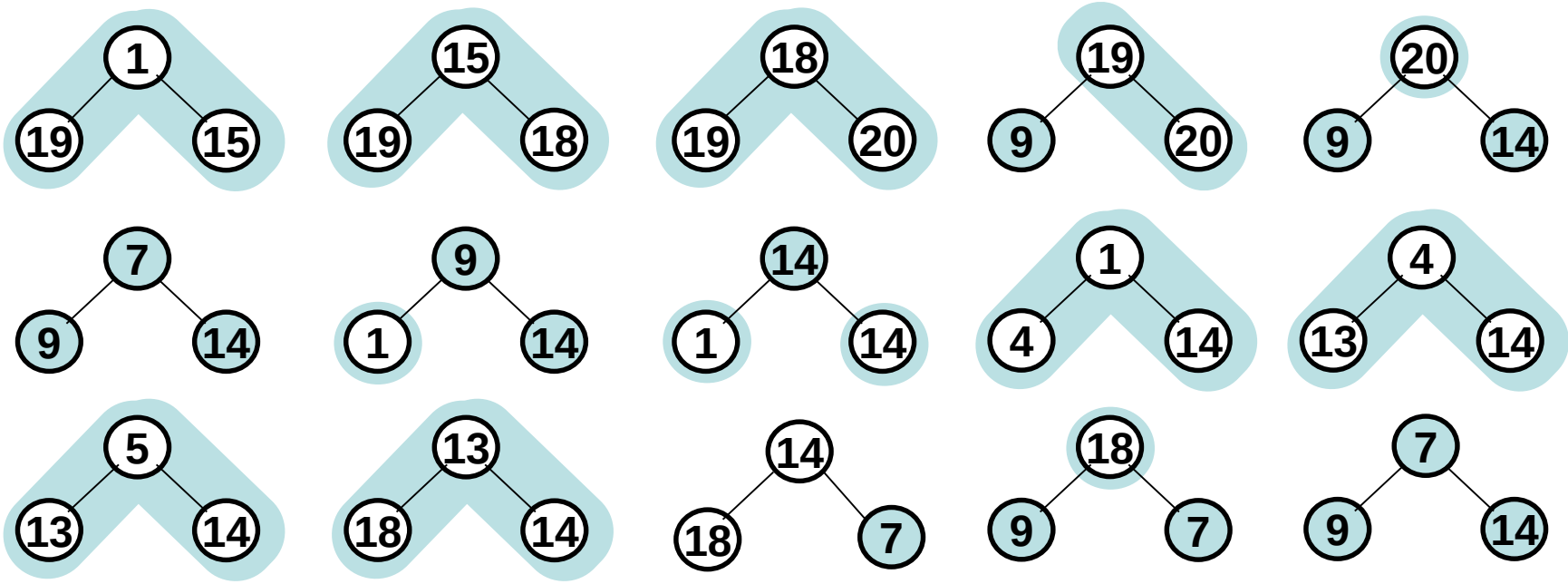
- 정렬의 첫 번째 단계에서 만들어지는 정렬된 블록이 크면 클수록 두 번째 단계에서 합병하는 횟수가 줄게 됨
- 이 때, 우선순위 큐를 사용하게 되면 첫 번째 단계에서 내부 메모리의 크기보다 긴 정렬된 블록을 생성할 수 있음
- 평균적으로 두 배 큰 블록을 생성할 수 있음

대치 선택(2)

- 정렬순서
  - 히프가 가득 찰 때까지 레코드를 히프에 삽입
  - 히프가 가득 차게 되면 히프로부터 하나씩 레코드를 삭제, 삭제한 레코드보다 새로 삽입되는 레코드의 키가 크면 동일한 정렬된 블록에 속함
  - 삭제한 레코드보다 새로 삽입되는 레코드의 키가 작으면 새로운 정렬된 블록에 속함, 이전 블록에 속한 레코드가 모두 없어질 때까지 히프에서 레코드를 삭제시킴

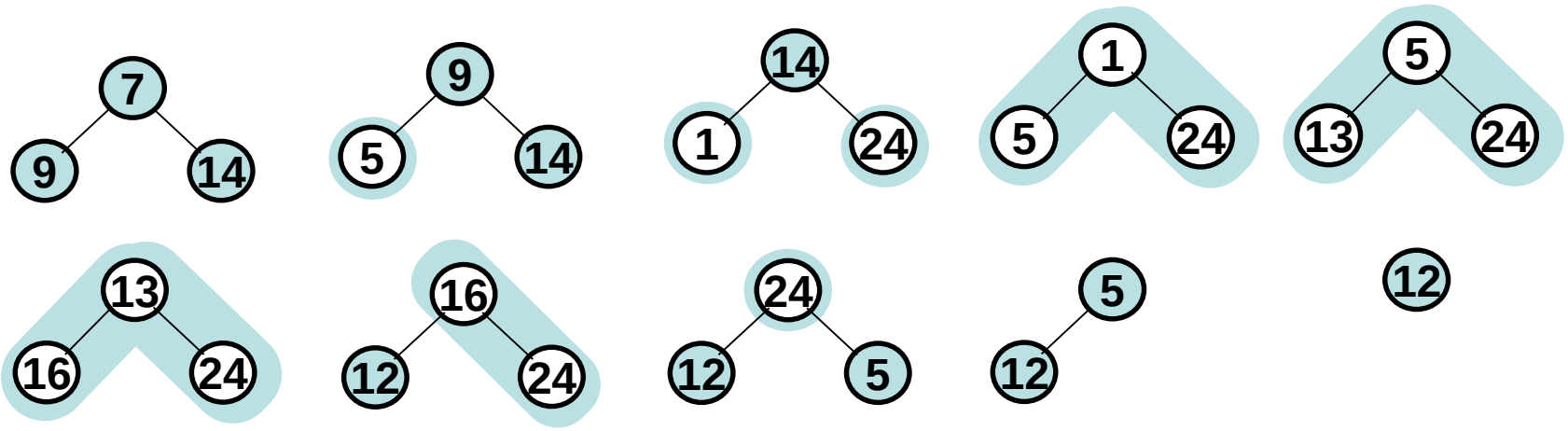
수행 과정(1)

- L = (1, 19, 15, 18, 20, 9, 14, 7, 1, 14, 4, 13, 5, 18, 7, 9, 14, …)



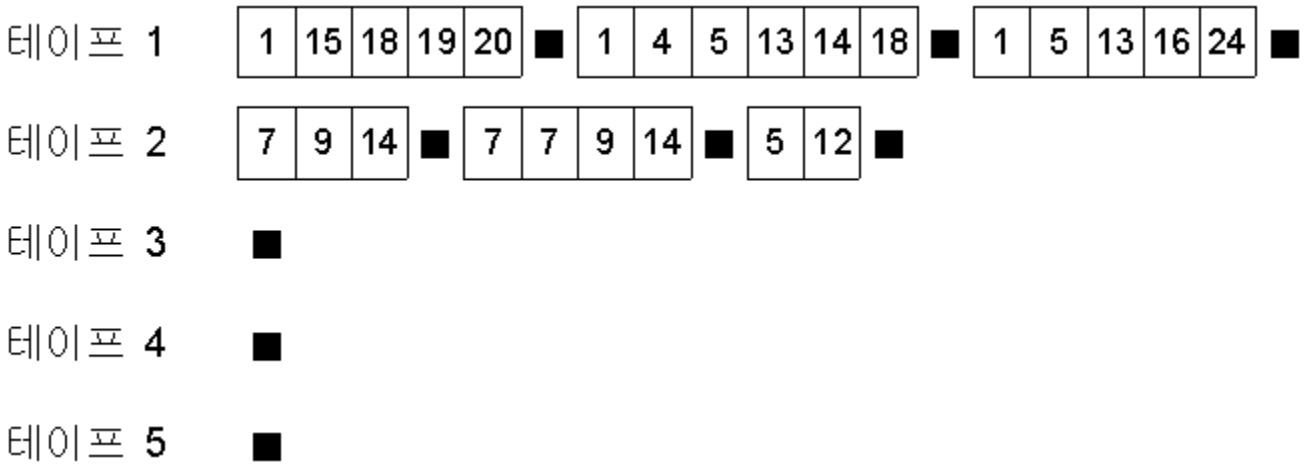
수행 과정(2)

- L = (…, 7, 5, 24, 1, 13, 16, 12, 5)



수행 과정(3)

- 첫 번째 블록 : 1, 15, 18, 19, 20
- 두 번째 블록 : 7, 9, 14
- 세 번째 블록 : 1, 4, 5, 13, 14, 18
- 네 번째 블록 : 7, 7, 9, 14
- 다섯 번째 블록 : 1, 5, 13, 16, 24
- 여섯 번째 블록 : 5, 12



다단계 합병 정렬

- 균형적 다방향 합병 정렬
  - 많은 수의 테이프와 많은 테이프 간의 복사가 필요
  - $2P$개의 테이프가 있을 경우 : 입력에 P개의 테이프를 사용하고 출력에 P개의 테이프를 사용
  - $P+1$개의 테이프가 있을 경우 : 합병 단계에서 하나의 출력 테이프와 P개의 입력 테이프 간의 복사가 발생 $\rightarrow$ $2P$개의 테이프가 있을 경우와 비교하여 두 배의 단계가 필요
- 다단계 합병 정렬
  - 하나의 테이프는 비워 놓은 상태에서 정렬된 블록을 고르지 않게 분산시킴
  - "공백까지 합병(merge-until-empty)" 전략을 사용

수행 과정(1)

- L = (1, 19, 15, 18, 20, 9, 14, 7, 1, 14, 4, 13, 5, 18, 7, 9, 14, 7, 5, 24, 1, 13, 16, 12, 5)

- 초기 정렬된 블록을 생성
  - 첫 번째 블록 : 1, 15, 18, 19, 20
  - 두 번째 블록 : 9, 14
  - 세 번째 블록 : 1, 7, 14
  - 네 번째 블록 : 4, 5, 13, 18
  - 다섯 번째 블록 : 7, 9, 14
  - 여섯 번째 블록 : 5, 7, 24
  - 일곱 번째 블록 : 1, 13, 16
  - 여덟 번째 블록 : 5, 12

- 테이프는 3개 사용

수행 과정(2)


테이프 1
1 15 18 19 20 ■ 9 14 ■ 1 7 14 ■ 4 5 13 18 ■ 7 9 14 ■
테이프 2
5 7 24 ■ 1 13 16 ■ 5 12 ■
테이프 3 ■
테이프 1
4 5 13 18 ■ 7 9 14 ■
테이프 2 ■
테이프 3
1 5 7 15 18 19 20 24 ■ 1 9 13 14 16 ■ 1 5 7 12 14 ■
테이프 1 ■
테이프 2
1 4 5 5 7 13 15 18 18 19 20 24 ■ 1 7 9 9 13 14 14 16 ■
테이프 3
1 5 7 12 14 ■

수행 예

- 다단계 합병 정렬을 위해 정렬된 블록을 분산시키는 예
- 테이프 6개 사용

테이프 1	*61*	*0*	*31*	*15*	*7*	*3*	*1*	*0*	*1*
테이프 2	*0*	*61*	*30*	*14*	*6*	*2*	*0*	*1*	*0*
테이프 3	*120*	*59*	*28*	*12*	*4*	*0*	*2*	*1*	*0*
테이프 4	*116*	*55*	*24*	*8*	*0*	*4*	*2*	*1*	*0*
테이프 5	*108*	*47*	*16*	*0*	*8*	*4*	*2*	*1*	*0*
테이프 6	*92*	*31*	*0*	*16*	*8*	*4*	*2*	*1*	*0*

성능 특성

- 다단계 합병 정렬은 P가 작을 경우에만 균형적 다방향 합병 정렬에 비해 좋은 성능을 가짐
- $P > 8$일 경우, 균형적 다방향 합병 정렬이 다단계 합병 정렬보다 빠르게 수행됨